SOFTWARE INSPECTION TOOL FOR DEFECTS CLASSIFICATION ON REQUIREMENT DOCUMENTS USING COLLABORATIVE APPROACH

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Sciences

By

Ishan Mani Subedi

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Program:
Software Engineering

May 2019

Fargo, North Dakota

North Dakota State University
Graduate School

Title

SOFTWARE INSPECTION TOOL FOR DEFECTS CLASSIFICATION ON REQUIREMENT DOCUMENTS USING COLLABORATIVE APPROACH

By

Ishan Mani Subedi

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Gursimran Walia
Chair

Dr. Pratap Kotala

Dr. Limin Zhang

Approved:

May 28, 2019
Date

Dr. Kendall Nygard
Department Chair

ABSTRACT

Software inspection is a process to minimize or eradicate defects throughout the entire software lifecycle which in turn helps to create and maintain quality products. While inspections are effective, it is often done manually and is time-consuming and tedious. To address this problem, this paper developed the "Software Inspection Tool" that can be used for identifying and detecting defects in software artefacts during the inspection process.

The tool is based on the Model(M), View(V) and Controller(C) architecture and uses modern web infrastructure like Amazon Web Services (AWS) for data storage, hosting and cloud computing. Representational State Transfer (REST) has been implemented to create an uncoupled system as well for third party information exchange.

Therefore, the creation of such a tool allows inspectors to collaborate globally to detect and report defects which further allows for creating quality software products and enable cost savings.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION AND BACKGROUND

Software inspection is a well-defined and rigorous review process where software artifacts are inspected in order to find and minimize defects (Fagan, 1986; Kalinowski and Travassos, 2004; Sauer et al., 2000).

Identification of defects in the earlier stages can decrease both cost and human effort when creating a software system. Fixing a problem in production is 100 % more costly than fixing it in the requirement or design phase (Boehm and Basili, 2001). Further 80% of the production bug fixing comes from 20 % of the initial defects (Boehm and Basili, 2001). Fagan, a prominent voice in the software inspection scholarships had done a role-based inspection process such as moderator, inspectors, author, and activities while a consistent taxonomy for the defects is influenced from the studies such as Alshazly et al. 2014; Teixeira et al., 2015 and Walia and Carver, 2009.

Over its lifecycle, a software system or a product can accumulate different defects types. As such dynamic or static analysis are necessary to remove the defects to minimize the cost of software products (Geraldi and Olivera Jr, 2017). In this paper, a software inspection tool (Hereby referred to as S.I.T) has been created which lets multiple inspectors collaborate and find defects in a software artefact.

Defect[s] is an umbrella term, which can have different meanings depending upon the context. A level of discipline should be applied to the word defect so that it is consistent among all inspectors and developers. Thus, we have a set of predefined defects which acts a yardstick and maintains consistency when inspectors are inspecting a particular software artefact (Alshazly et al., 2014).

A study (Fagan, 2002) shows that Omission, Incorrect Facts, Inconsistency, Ambiguities and Extraneous information to be the major types of predefined defects. Geraldi and Oliveira Jr in their paper "Defect Types and Software Inspection Techniques: A Systematic Mapping Study" define the defects in the following way:

Omissions: the absence of a mandatory element or functionality, e.g., variations of a certain requirement not present in the specification.

Incorrect Facts: a functional requirement or use cases incorrectly described

Inconsistencies: problems from functional requirements or use cases with their goals and specifications poorly designed - e.g., descriptions, variations and, terminology.

Ambiguities: A specified functional requirement or use case that does not meet its objective - e.g., descriptions with multiple interpretations or misdescribed

Extraneous Information: functional requirements or use cases are redundant - e.g., are duplicated and without specification.

Using the S.I.T, inspectors can collaborate and adhere to the above defect criteria while creating a detailed report.

2. FEATURES

Software inspector product is focused on reducing potential defects by identifying them at the requirements phase. After the necessary requirements have been gathered from clients and stakeholders, the review team can use the software inspector tool to collaboratively find flaws.

The actual tool developed for this project can be categorized into two different sections: users and admin.

Here are the list of things an admin can perform on the platform:

- Admin can add inspection documents from the admin panel
- Admin can manage users from the admin panel
- Managing users include adding new users or deleting existing users
- Managing users also include editing name, emails, and platform access permissions of the existing users
- Admin can add collaborators to the inspection documents from the admin panel
- Admin can see recent activities happening on the platform from the admin panel
- On the platform, admin can see

Here are the list of things users can perform on the platform:

- Users can view the documents uploaded by admin to which they were invited as a collaborator
- Users can open the documents and identify defects within different pages by filling out a row-wise forms along with adding a time and date stamp
- Users can select from following defects types: Omission, Ambiguous Information, Inconsistent Information, Incorrect Fact, Extraneous and Miscellaneous
- Users can add or delete new row to fill out the defect information

3. IMPLEMENTATION OF THE SOFTWARE INSPECTION TOOL

3.1. Model View Controller

The software inspection tool is based on the MVC or the Model-View-Controller architecture. In this type of architecture, the model contains the data layer while the controller layer takes the request and routes it to the view which in turn shows the information to the user. As the web is basically a series of HTTP request this model is quite suitable for the web.

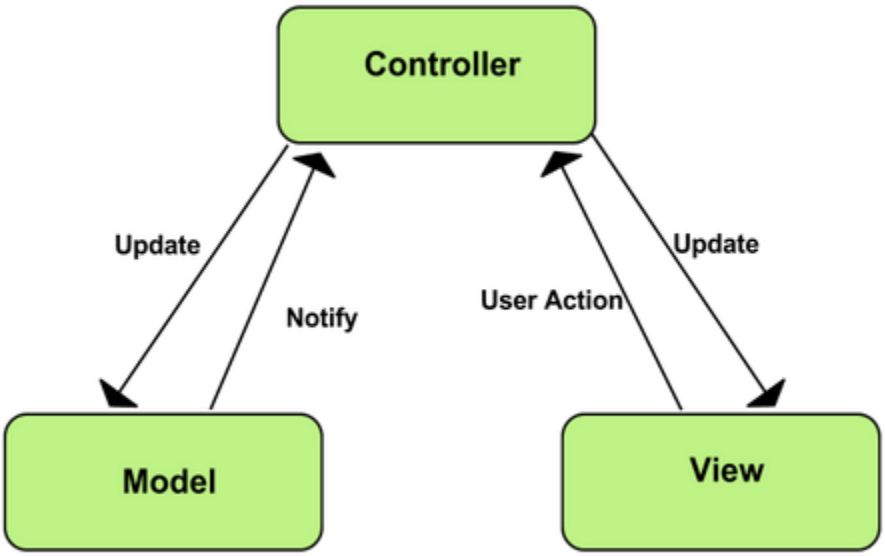


Figure 1: MVC architecture

The software inspector tool uses Django, a high-level Python web framework which is also based on the MVC architecture albeit with a little tweak. Django likes to call its architecture MVT i.e Model-View-Template ("Django" ,2018). The Model still holds the data and sends it to the view via URL controller, but the view injects data into the template which is then shown to the user. Django uses the Django Template Language (DTL) to handle injected data. Let's dive deeper into the next section to see the Models, Views and Controllers in context to the software inspection tool.

3.2. Models

Models are the single, definitive source of information about your data ("Models, 2018"). The models define the fields with their respective types and behaviors. Models also have database-abstraction API that lets you create, retrieve, update and delete objects("Database",2018). In S.I.T we have taken code first approach, as such the database tables are generated from the code rather than directly using SQL to define database tables. Readers should be aware that this does not mean we are not using SQL. The ORM (object-relational mapper) does this for us as per the code basis("Database",2018) The following is an example of a model taken directly from the S.I.T

```
class Userprofile(models.Model):
   user = models.ForeignKey(settings.AUTH_USER_MODEL)
   fault = jsonfield.JSONField()
inspectionDocument=models.ForeignKey(InspectionDocuments, null=False)
```

3.3. Views

Views in the S.I.T work through a request and response cycle. Each view gets a web request and then returns a web response depending upon that request ("Writing Views",2018). A request is a python object that contains metadata while the response could be anything from an HTML page, image or even JSON. The following is an example of a view taken from the S.I.T which shows the dashboard view protected by the @staff_member_required decorator.

```
@staff_member_required
def dashboard(request):
   all_queryset = CollectionProfiles.objects.all()
 return render (request,'list-of-dashboard-items.html',
{'userObjects':all_queryset})
```

3.4. Template

The template is what is shown to the end-user. Let's look at the dashboard view again to see how a template really works.

```
@staff_member_required
def dashboard(request):
   all_queryset = CollectionProfiles.objects.all()
return render(request,'list-of-dashboard-items.html',{'userObjects':all_queryset})
```

As the readers can see the *userObjects* is being injected into the 'list-of-dashboard-items.html' template which is then presented to the user. The template supports HTML, CSS and JavaScript and shows the result to the user.

3.5. Class Diagram

The class diagram shows the relationship between all the classes that are used in the software inspector tool. Let's take a look at them one by one.

Abstract Base User: This is an abstract base class from which the User class is derived from.

User Class: The User class is derived from the Abstract base user class. The user class is responsible for signing up a user, logging in, confirming the user and logging out the user.

Confirm User class: In the S.I.T the admin has to approve each inspector that signs in into the platform. Without the admin approving of the process, the inspector cannot log into the portal. This is by design as the admin has to check the credential of the inspector that is logging into the portal.

Inspection Documents class: The inspection document class takes care of uploading the requirements document that needs to be inspected by the inspector. The documents can only be uploaded by the person who has sufficient privilege.

Collection Profile class: The collection profile class is responsible for collecting and maintaining whatever comments are given to particular inspection documents.

Userprofile class: The userprofile class is responsible for collecting and maintaining whatever comments are given by the individual user for a single requirement document. Unlike the collection profile class which maintains the record about a particular inspection document, the userprofile class maintains the record about the comments given by an inspector.

Permission class: As the name suggests the permission class is responsible for giving privileges. The privileges can be at various levels from uploading the document to becoming an inspector who can look and comment only for a particular document.

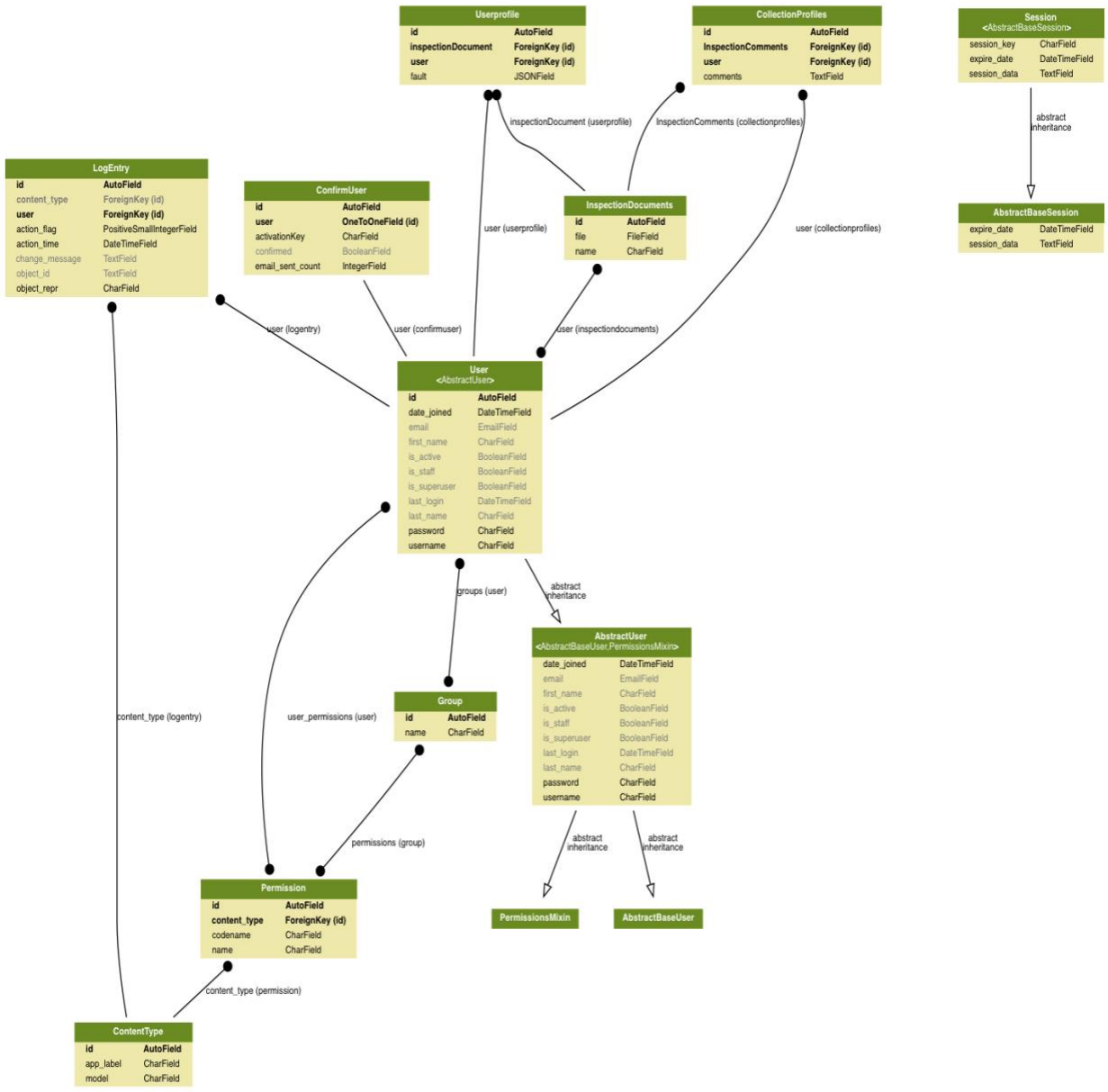


Figure 2: Application diagram

3.6. RESTful API (Representational State Transfer)

"The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system"(Fielding,2000). With the arrival of smartphones, tablets and other portable gadgets, REST has become a necessity for the modern web architecture because it allows the backend system to be uncoupled with the front-end

system. It also allows third parties to consume endpoints/ API (Application programming interface) to make their own applications. For example, Google Maps a really popular GPS tool has exposed its REStFul services so that other developers can consume the information for making other applications. The architecture of the S.I.T is not in Pure REST. This would mean that the backend system is in isolation with the frontend system. However, the S.I.T tool exposes the information about required documents through its RESTful services given that the client has sufficient privilege to do so. The modern web development advocates JSON (JavaScript Object Notation) and XML (extended hyperlink language) as the primary protocols for information transfer. The S.I.T uses JSON for information transfer because it is more suited for modern frameworks like Angular, React native, React JS for parsing and consumption.

3.7. Task Queuing

A particular requirement document can be accessed by more than one inspector at a particular time. This means that more than one inspector can access, comment and save their comments on a particular requirement document. As such we do not want to allow any deadlocks and want to keep the comments of each inspector in isolation. Hence S.I.T uses tools like Celery and Redis to queue tasks which blocks any process to be overlapped and enables processes to be processed in the first-come, first-serve basis following the classical queue data structure. Celery is an asynchronous task queue which is based on distributed message passing while Redis is an in-memory data structure store, used as a database, cache and message broker.

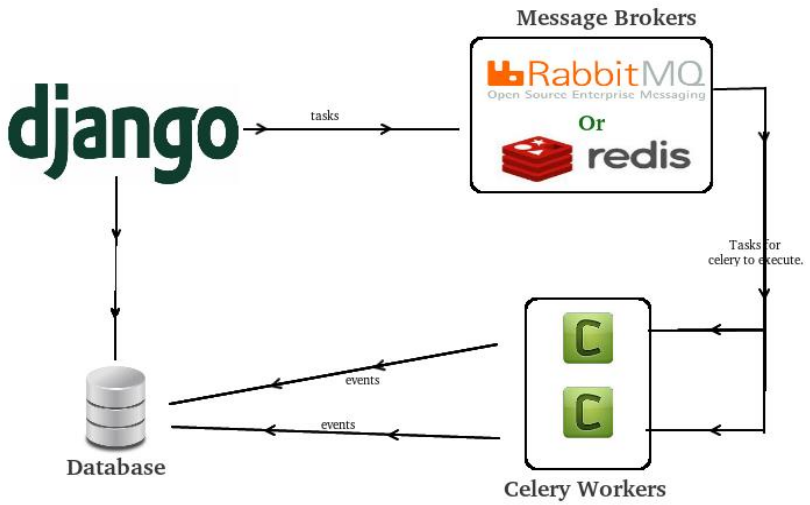


Figure 3: Task queue diagram
Source: https://en.proft.me/2013/10/25/celery-periodic-tasks-django-projects/

3.8. The Infrastructure of the Software Inspection Tool

The infrastructure of the S.I.T follows that of the modern industry standard. The backend code is written in Django and the frontend code is written using HTML, CSS and a simple responsive Bootstrap framework. The code has been deployed in Heroku which is the cloud server similar to the popular Amazon Web Services. The database of the S.I.T is in PostgreSQL because the database drivers and tools are very compatible with Django. In the S.I.T the Requirement documents that are to be inspected are not stored in the database. In the olden days, the static documents like such would be converted in base64 and would be chunked into several records which would then be stored in the database. This would sometime create an inconsistency when converting back to the original form. Since the requirement documents are sensitive, the S.I.T tool stores the actual document in the S3 bucket. Amazon S3 is cloud storage for the internet which lets you store any type of data ranging from photo, video, document and also provide a plethora of rich Apis to manage them ("Amazon S3 Rest API Introduction",2018).

Hence only a pointer to the document would be set in the database. This creates an isolated, uncoupled system which is always desired.

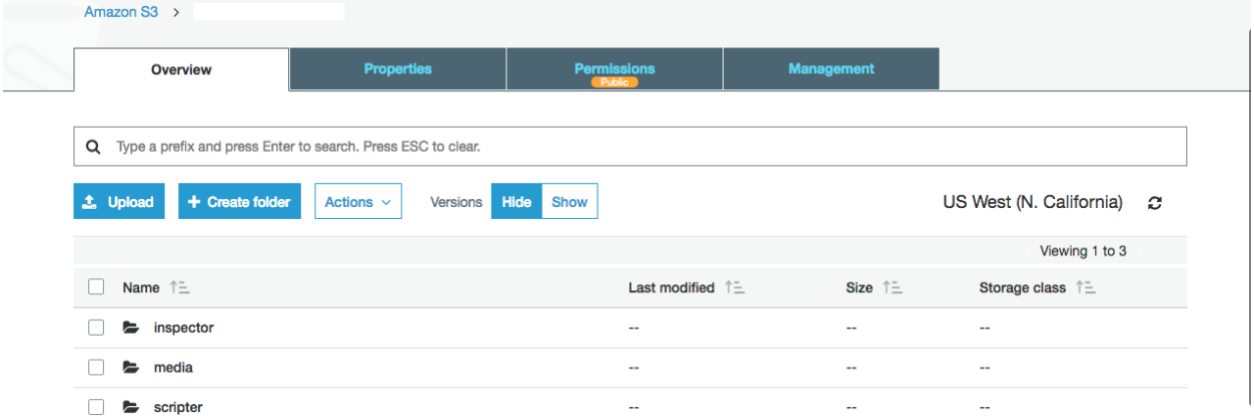


Figure 4: The AWS bucket for file storage

The S.I.T uses git to do its version control. The repository is maintained in bitbucket and all and any changes can be pushed to the cloud server via a continuous integration.

3.9. Overview of the S.I.T tool

The software inspector tool has two levels of access. The first level of access is the superuser access while the second level of access is the inspector access. We shall look at them one by one.

3.9.1. The superuser access

The superuser access is the highest level of access in the software inspector tool. The superuser has the ability to upload documents, confirm to deny access to any existing inspectors and grant access to new inspectors who are willing to inspect any requirement documents. The superuser also decides which inspector has access to which requirement document.

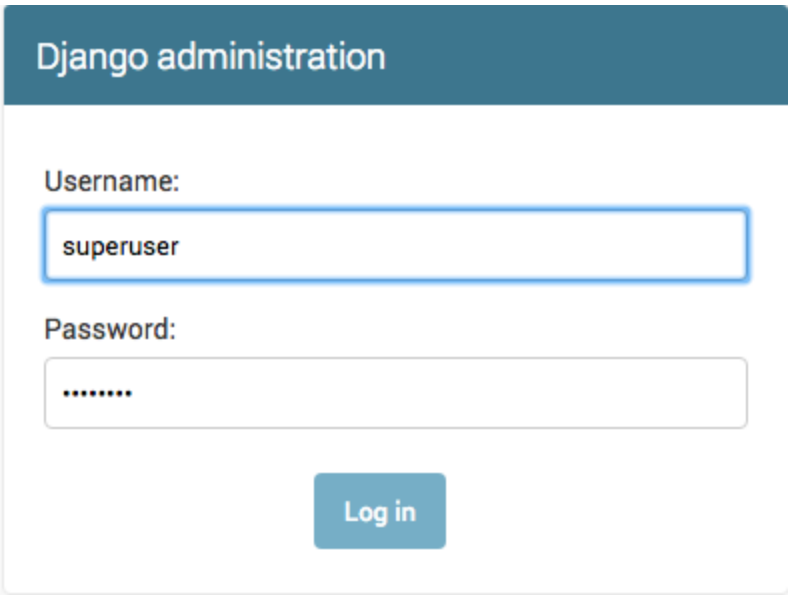


Figure 5: Superuser login page

Fig 4 shows the superuser login page. This page can be accessed by the https://software-inspector.herokuapp.com/admin link. The reader should keep in mind that although the domain name is subject to change in the future the /admin link would remain the same. Only the superuser can log in to this page. Incorrect login will yield the result like in fig. and successful login will yield the result in fig 6

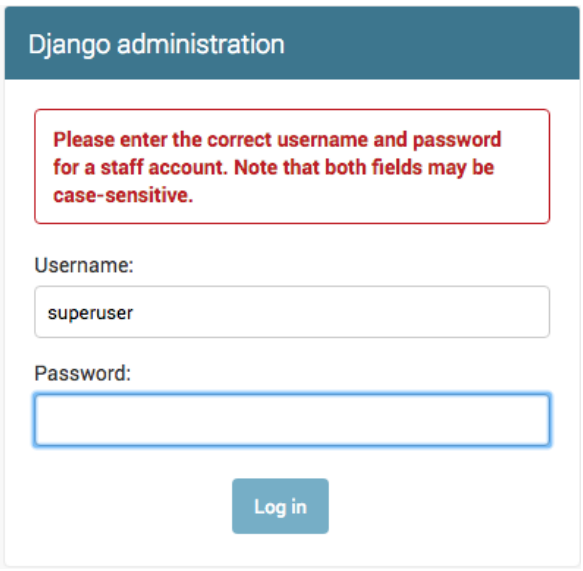


Figure 6: Incorrect superuser login

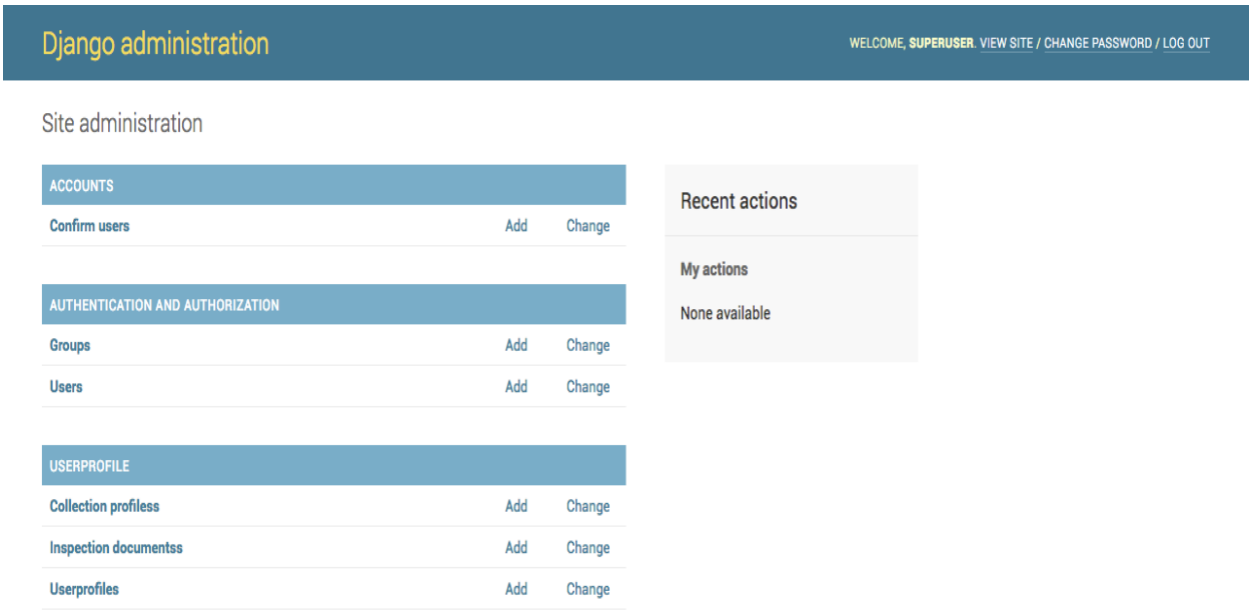


Figure 7: Superuser portal

Fig shows the portal presented to the superuser. The superuser has complete privilege over all the information that is available in S.I.T tool. As stated above the superuser has the ability to confirm new users, create groups, add or remove users, look at the comments and information pertaining to certain documents.

3.9.2. Confirm user flow

Whenever new users sign in the portal the superuser has to give them access so that they can go ahead and inspect the requirement document. Without the superuser giving them the green light users cannot go and sign in.

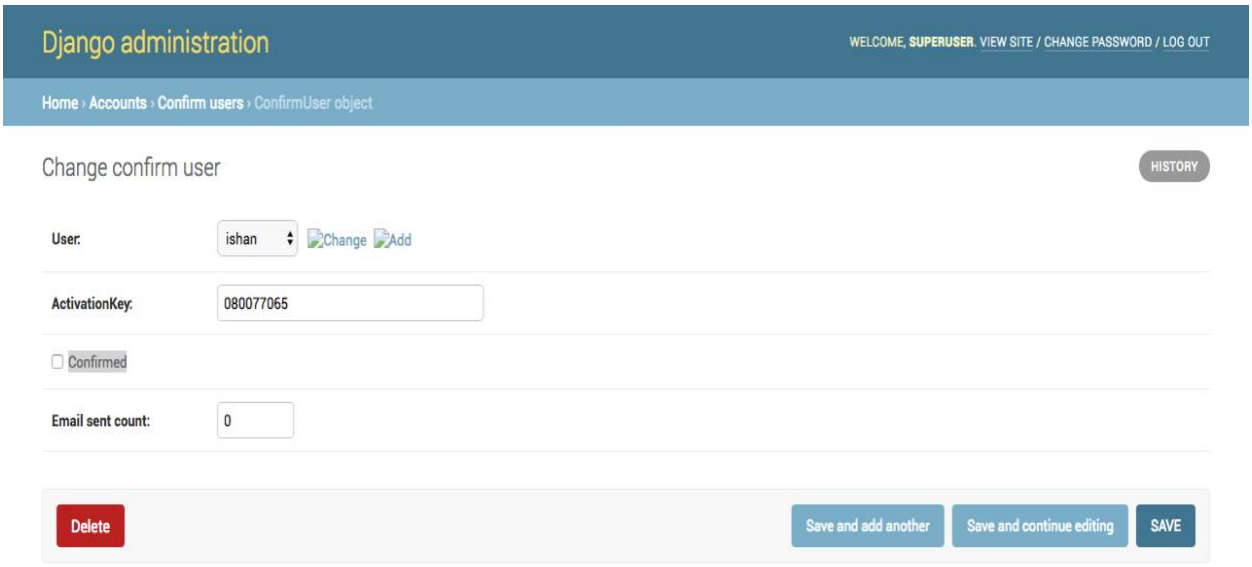


Figure 8: The user confirmation portal

As shown in Fig 8 the user Ishan has to be confirmed by the superuser before gaining access to the portal.

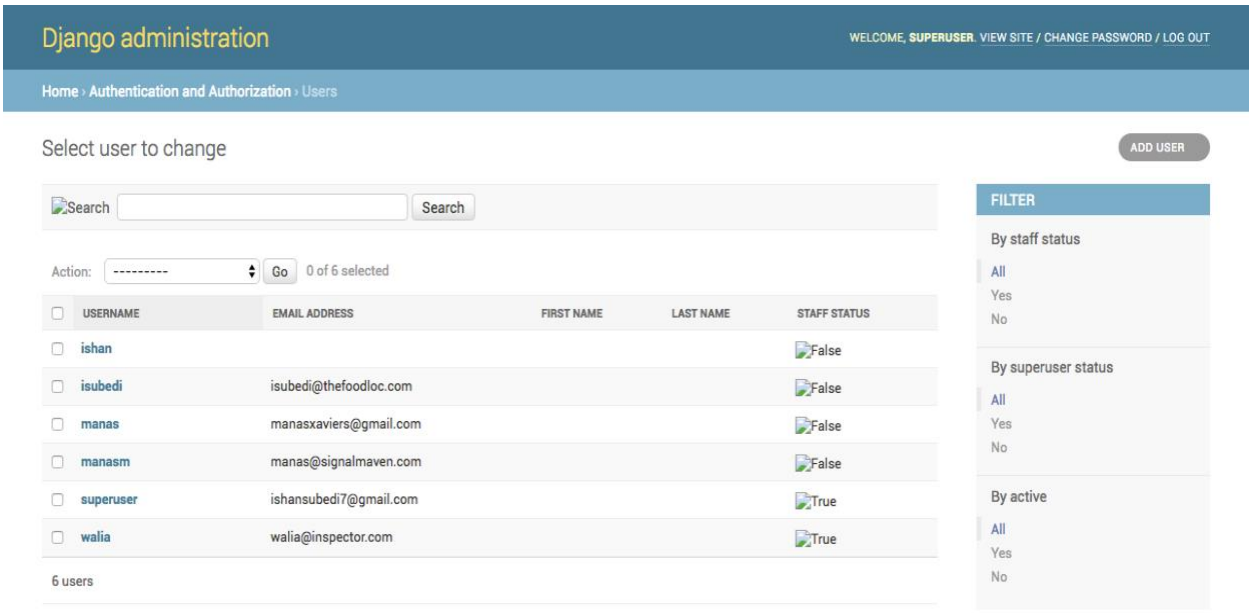


Figure 9: List of inspectors

3.9.3. Inspector roster

As shown in the figure above the superuser has the ability to look at the list of inspectors who are currently in the S.I.T Tool. The tool also gives them privileges to delete or edit any user information.

3.9.4. Collection profile

The collection profile gives the superuser to view the comments and suggestions as given to a particular requirement in JSON format. The comments are not necessarily meant for human reading it but are meant for storage. These comments are nicely parsed and presented in a human-readable form as we can see in the later chapters. The collection profile is shown in figure.

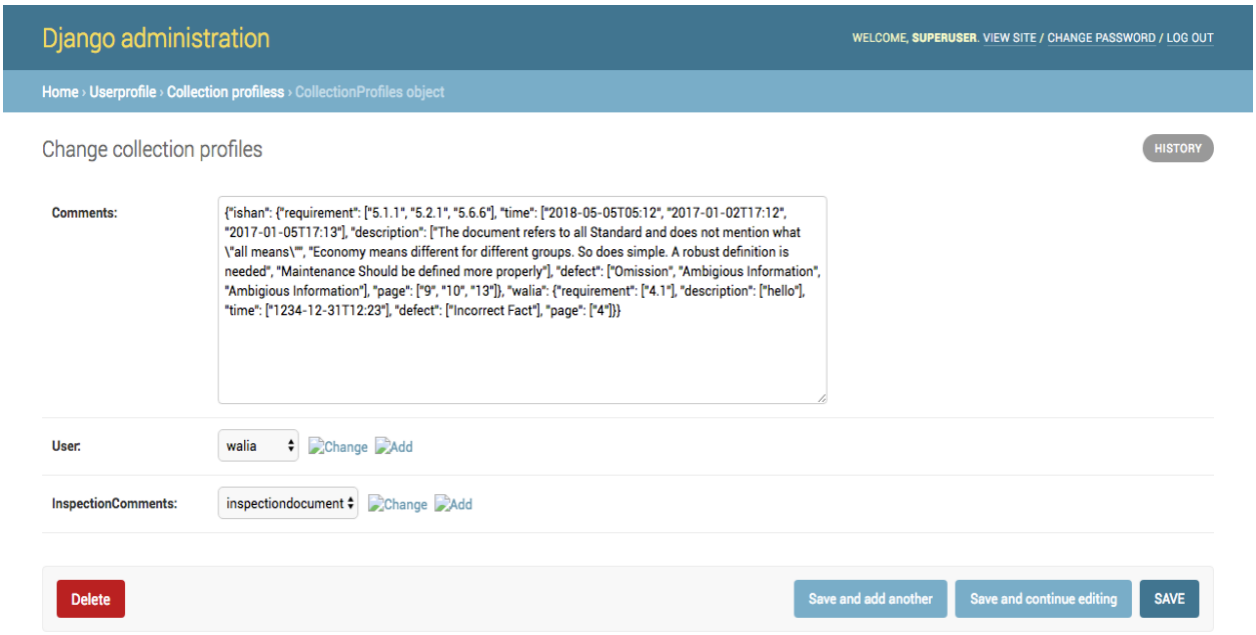


Figure 10: The collection profile

3.9.5. Inspection documents

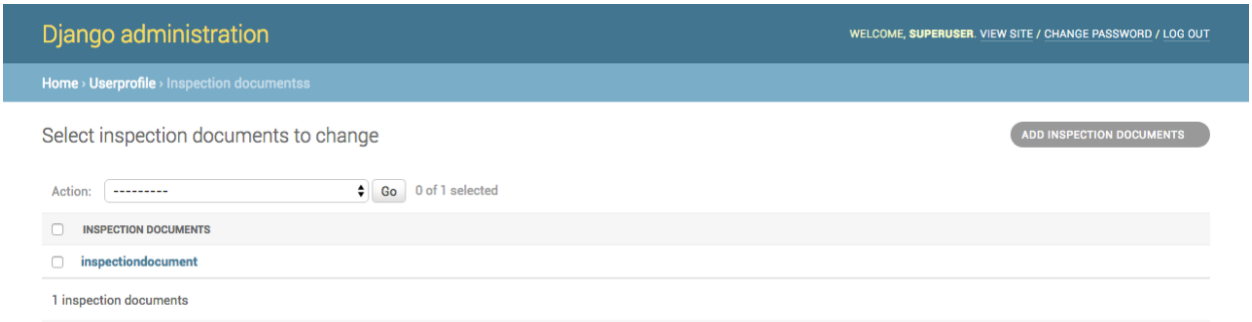


Figure 11: The requirement portal

After clicking on the add inspection document, the superuser can select which document to upload as shown in Fig 12

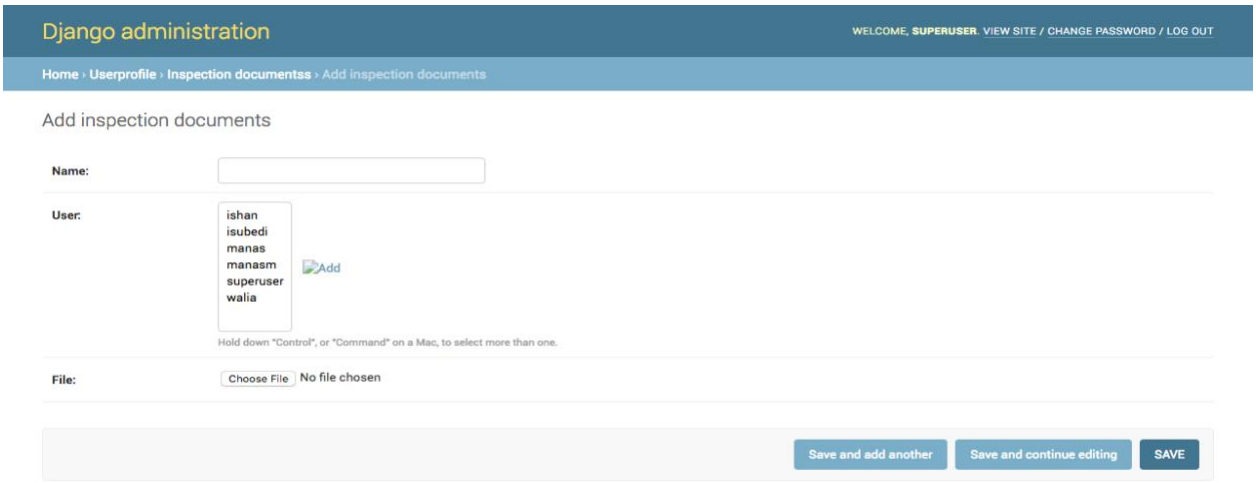


Figure 12: The requirement document portal

As shown in the figure above the superuser has the ability to upload requirement documents. The superuser would simply click on choose file button and upload the required documents. And also select which inspectors (shown as users in the figure) to give access to the uploaded document. Readers should also note that the inspection document name should be unique, otherwise the tool will throw an error complaining of the duplicate name.

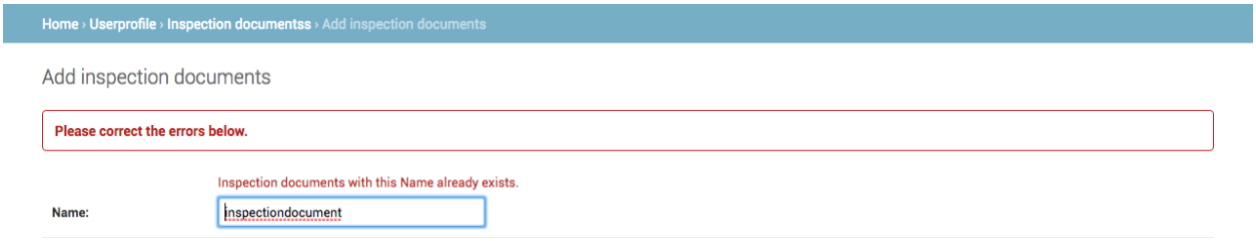


Figure 13: Document duplicate entry

3.9.6. Userprofile

As the name suggests the Userprofile shows the comments given by a certain inspector pertaining to a certain document. While the collection profile keeps the record about a document the userprofile keeps the record about a certain user. The data stored is in JSON Format. The comments are not necessarily meant for human reading it but are meant for storage.

These comments are nicely parsed and presented in a human-readable form as we can see in the later chapters. The collection profile is shown in figure 13.

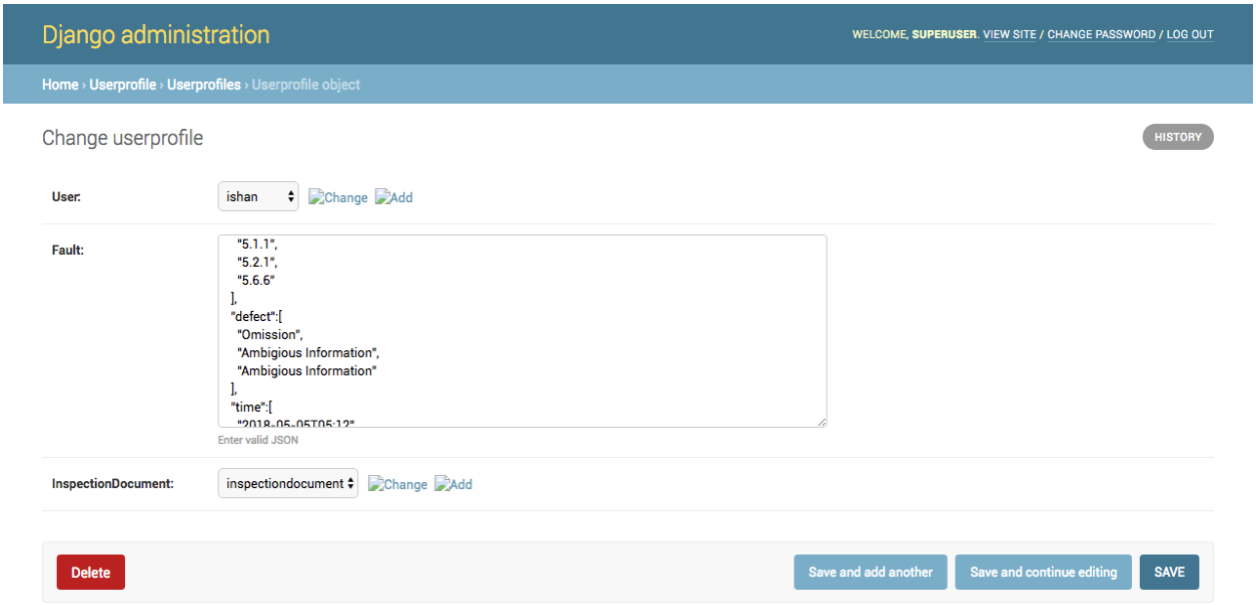


Figure 14: Userprofile page

As the figure shows the contains the information given by a certain user to a certain document. In this case it shows ishan's comment on a document called inspection Document.

3.9.7. The inspector access

Readers should note that every superuser is also an inspector, but the reverse is not correct. Hence the superuser also enjoys the privilege of being an inspector. Let's start from the inspector signup process and dive into the inspector portal. The S.I.T tool can be accessed via the https://software-inspector.herokuapp.com/

Figure 15: Homepage of S.I.T

3.9.8. Signup

Anyone willing to become an inspector would click on the top right corner and click on signup which would show the figure below.

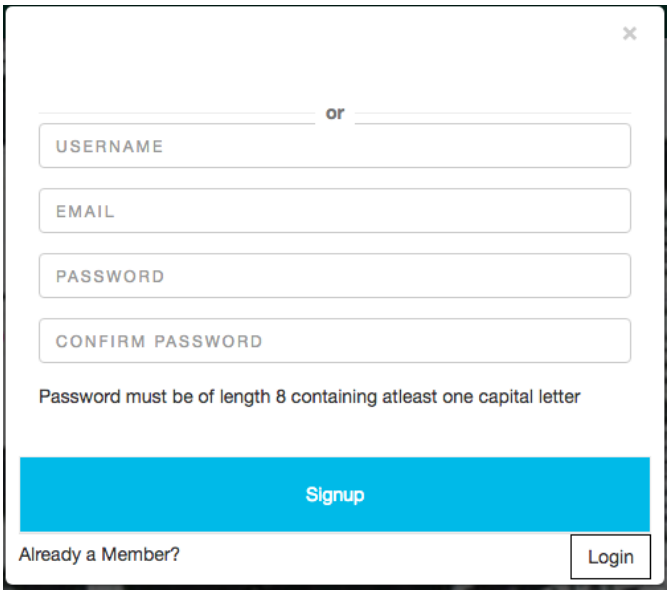


Figure 16: Signup page

As readers can see anyone willing to become an inspector needs to select a username, email and password. The S.I.T will complain in case of any discrepancies or give a success message upon signup completion both of which are shown in Fig 15 and Fig 16.

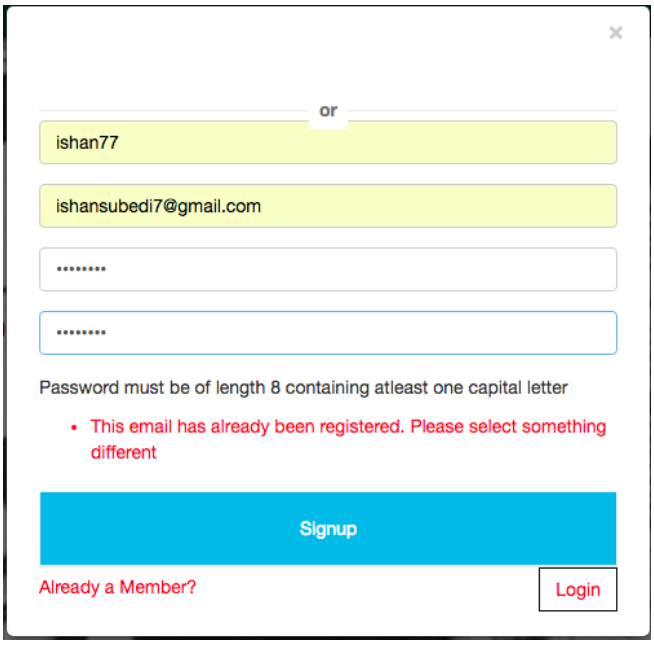


Figure 17: Signup error

As shown in Fig the tool complains in case of discrepancies. In this case the email was duplicate and was already in the system. The tool will complain even for duplicate usernames and password that do not follow the regex format of "^(?=.*?[A-Z]).*\d" which means that the password should contain at least one capital letter. The password length should be more than eight characters. If the form has been filled correctly a successful signup will yield the following screen.

Thank You!

Please check your email for further instructions on how to complete your account setup.

Figure 18: Successful signup process

The tool will now send an email to the user stating that it has received the information and the user is in still in the verification process. If the inspector tries to login while still in verification, the inspector is presented with figure 19

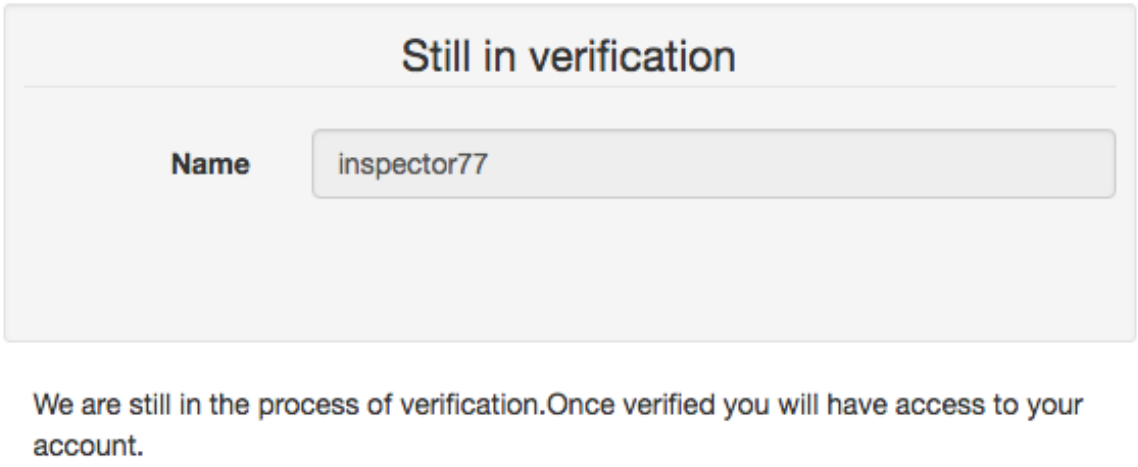


Figure 19: Still in verification

At this point the superuser has to approve the user/inspector to get access. Readers may refer to Fig 19 on what this means. After the user is approved by the super user the inspector finally gets access to access the portal as shown in Fig 20

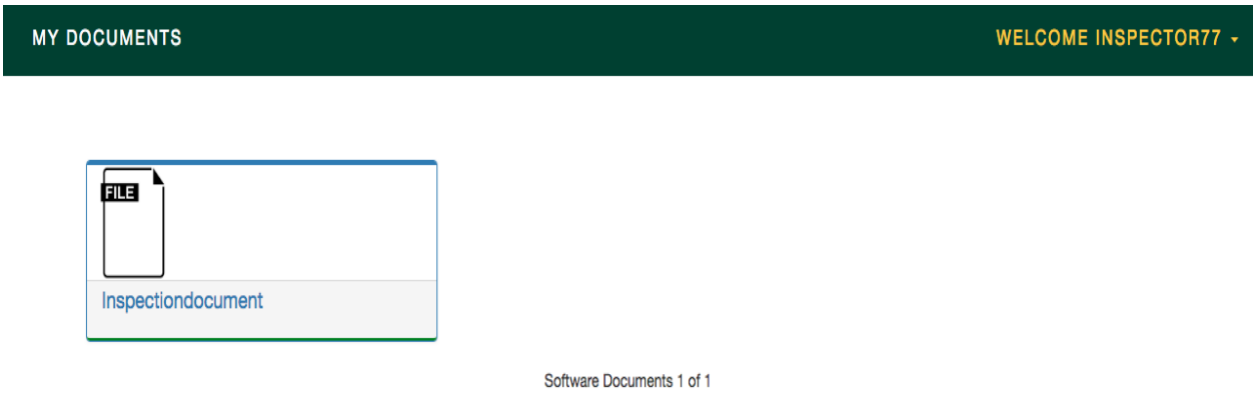


Figure 20: The inspector portal

Fig 20 shows the inspector portal. Here for our dummy user "inspector77" the super user has given access to the inspection document. In other words, inspector now has the ability to make checks and comments on the given requirement document.

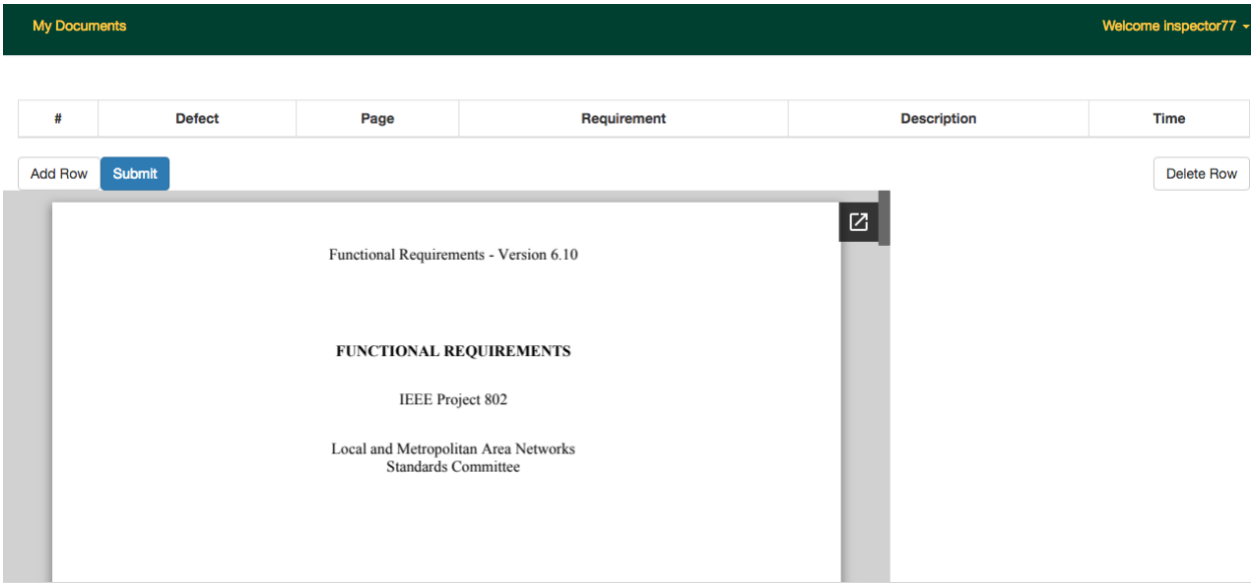


Figure 21: Document portal

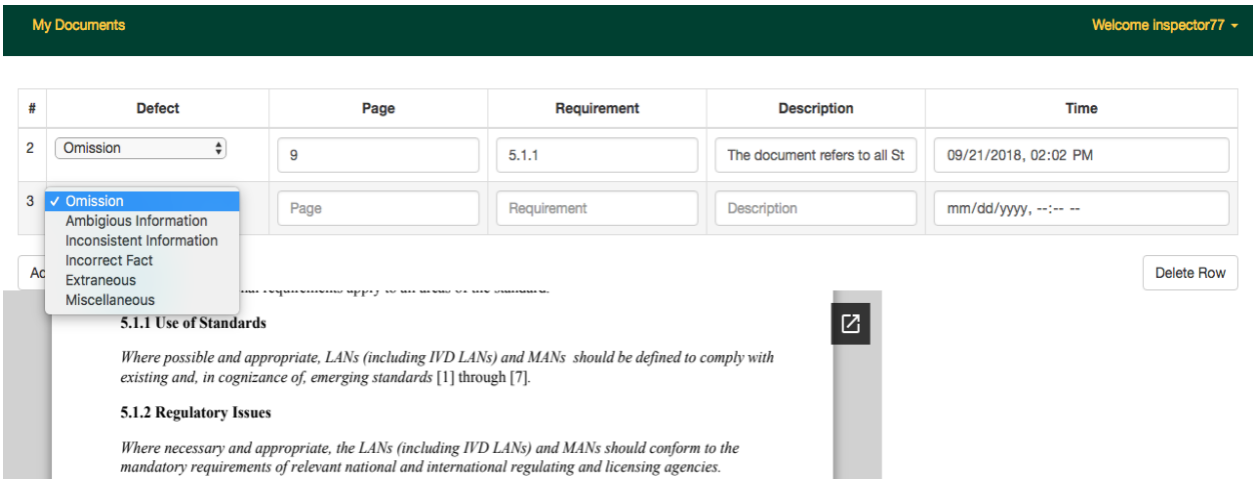


Figure 22: Inspector adding comments

The above figure shows how an inspector can choose from Omission, Ambiguous Information, Inconsistent Information, Incorrect Fact, Extraneous or Miscellaneous. Then the inspector can fill in the page, requirement number with description and time. Any number of rows can be added and deleted. After the inspector is done, they can hit submit which will save their progress.

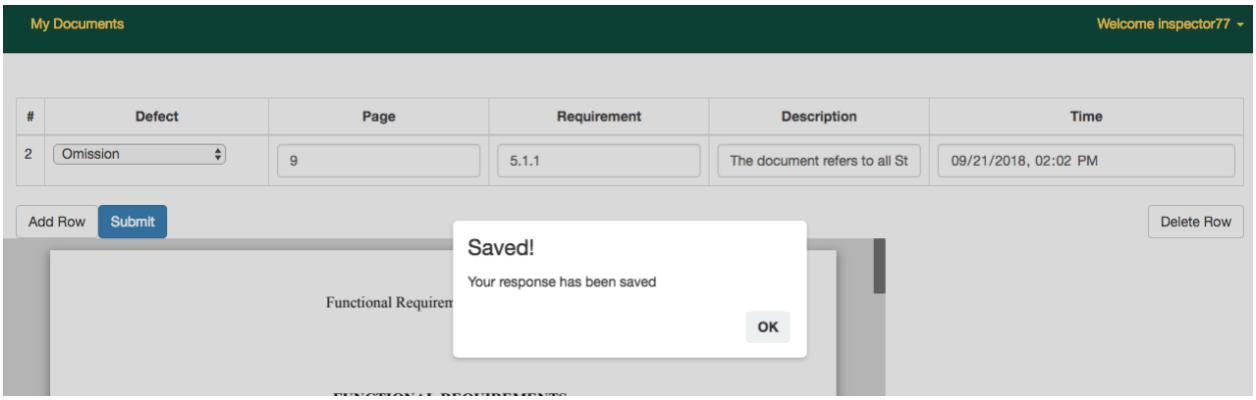


Figure 23: Inspector saving their response

As shown in the figure the inspector can submit by clicking the submit button which does an ajax call and saves the response. The inspector however does not have the capability to look at the comments of other inspectors. This is intentional as to reduce the bias and keep the

originality. The superuser however has that capability and can look at the comments via their personal dashboard.

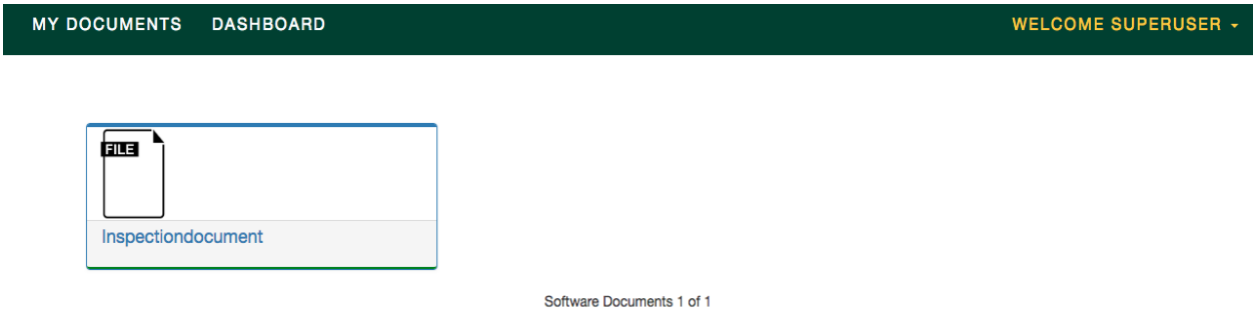
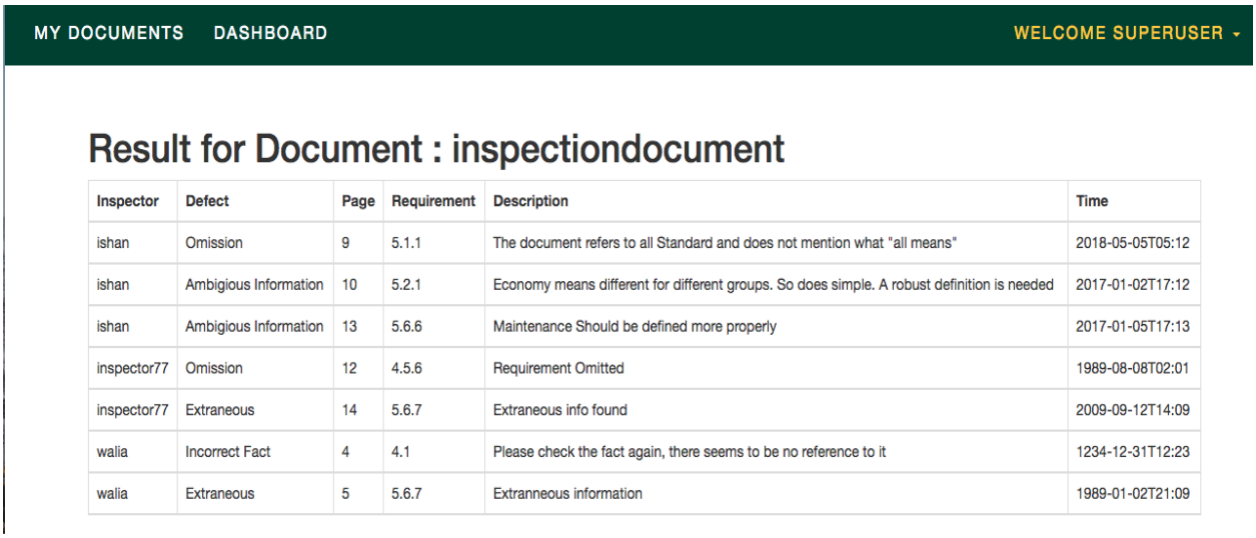


Figure 24: The superuser dashboard

The figure above shows the superuser dashboard. Readers should note that the Dashboard navigation menu in the top right is only available for the superuser. The Dashboard will show all the information related to the inspection document. Please refer to the figure 25



Inspector	Defect	Page	Requirement	Description	Time
ishan	Omission	9	5.1.1	The document refers to all Standard and does not mention what "all means"	2018-05-05T05:12
ishan	Ambigious Information	10	5.2.1	Economy means different for different groups. So does simple. A robust definition is needed	2017-01-02T17:12
ishan	Ambigious Information	13	5.6.6	Maintenance Should be defined more properly	2017-01-05T17:13
inspector77	Omission	12	4.5.6	Requirement Omitted	1989-08-08T02:01
inspector77	Extraneous	14	5.6.7	Extraneous info found	2009-09-12T14:09
walia	Incorrect Fact	4	4.1	Please check the fact again, there seems to be no reference to it	1234-12-31T12:23
walia	Extraneous	5	5.6.7	Extranneous information	1989-01-02T21:09

Figure 25: Displays comments

Rest API for demonstration purpose: The S.I.T tools as stated in the previous chapters expose a rest endpoint, just for demonstration purposes on how the API of the S.I.T could be consumed by other devices. The REST endpoint is exposed at https://software-inspector.herokuapp.com/api/show-comments/?document-name=inspectiondocument. The query

string document-name = inspectiondocument refers to the name of the requirement document whose information end users want to see. Calling that endpoint would give the following result:

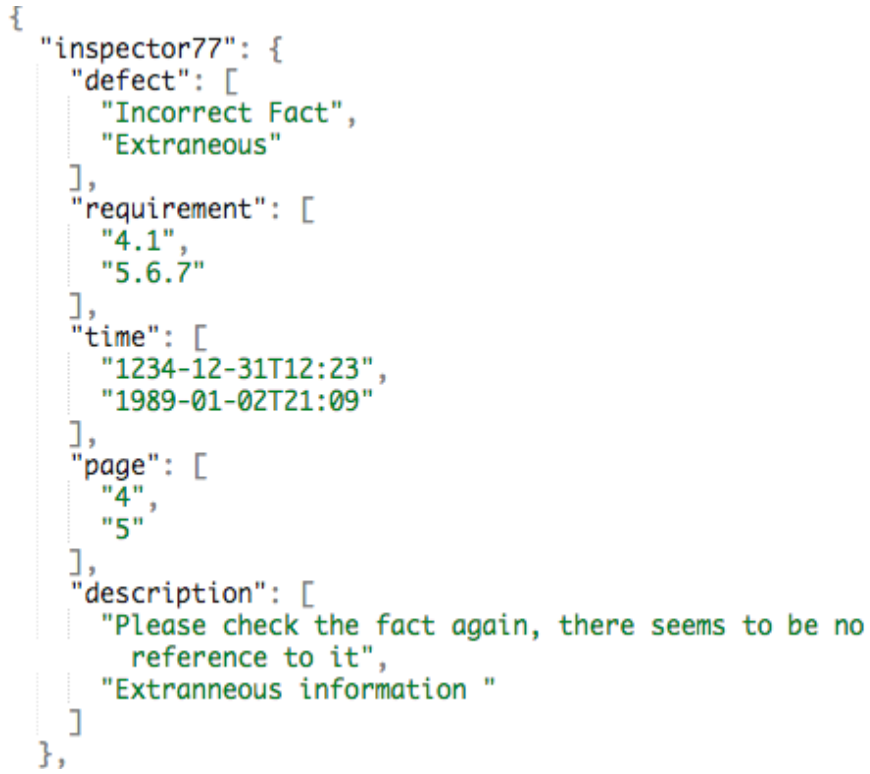


Figure 26: Calling JSON API

Since all the modern browsers and frameworks can parse JSON, any mobile, tablet devices can consume this API to create an interface. If the document does not exist, the API will simply throw an error as: {"Error": "The requested document was not found please check the name and try again."}

4. CONCLUSION AND FUTURE WORK

This paper explores the S.I.T in its prototype phase. As such students could use the tool for a software requirement (or any other for that matter) class and give feedback for further improvement. S.I.T could use LEAN (Ries, 2011) principles where a product is quickly given to a client and feedback is filtered and applied using Agile methodologies.

The tool has been written by using scalable architecture and APIs hence the architecture is very scalable in case the system sees high user signups and even higher traffic.

The tool is in its prototype phase; hence it has been written as a platform to test our proof of concept about discovering defects in software artifacts. Therefore, the level of machine learning and automation has been intentionally minimized. With ample resources automatic defect detection by using machine learning algorithms, dynamic pdf editing, intext highlighting and dynamic question- answer sections could be incorporated. The S.I.T could also greatly benefit from the data analysis. By incorporating graphs and charts readers can see defect patterns that are occurring per artifact basis.

As the foundation has already been written using REST, the S.I.T would highly benefit from a consistent user interface which is mobile, desktop and tablet friendly. Albeit, some level of responsiveness has been added a lot more could be incorporated.

REFERENCES

Alshazly, A.A., A.M. El Fatatry and M.S. Abougabal, 2014. Detecting defects in software requirements specification. Alexandria Eng. J., 53: 513-527. DOI: 10.1016/j.aej.2014.06.001

Boehm, B. and V.R. Basili, 2001. Software defect reduction top 10 list. J. Comput., 34: 135-137. DOI: 10.1109/2.962984

Database. 2018, May 27. Retrieved November 5, 2018, from https://docs.djangoproject.com/en/2.1/ref/databases/

Django (Version 2.1) [Computer Software]. 2018. Retrieved from https://djangoproject.com

Fagan, M., 2002. A History of Software Inspections. 1st Edn., Springer, Berlin Heidelberg.

Fagan, M., 2002. A History of Software Inspections. 1st Edn., Springer, Berlin Heidelberg.

Fielding, R. T., & Taylor, R. N. 2002. Principled design of the modern Web architecture. ACM Transactions on Internet Technology (TOIT), 2(2), 115-150.

Kalinowski, M. and G.H. Travassos, 2004. A computational framework for supporting software inspections. Proceedings of the 19th International Conference on Automated Software Engineering, Sep. 24-24, IEEE Xplore Press, Linz, Austria, pp: 46-55, DOI: 10.1109/ASE.2004.1342723

Kalinowski, M. and G.H. Travassos, 2004. A computational framework for supporting software inspections. Proceedings of the 19th International Conference on Automated Software Engineering, Sep. 24-24, IEEE Xplore Press, Linz, Austria, pp: 46-55, DOI: 10.1109/ASE.2004.1342723

Models. 2018, May 27. Retrieved November 5 2018, from https://docs.djangoproject.com/en/2.0/topics/db/models/

Object Storage Features – Amazon Simple Storage Service (S3) – AWS. (n.d.). Retrieved from https://aws.amazon.com/s3/features/

Sauer, C., D.R. Jeffery, L. Land and P. Yetton, 2000. The effectiveness of software development technical reviews: A behaviorally motivated program of research. IEEE Trans. Software Eng. 26: 1-14. DOI 10.1109/32.825763

Silva, N. and M. Vieira, 2016. Software for embedded systems: a quality assessment based on improved odc taxonomy. Proceedings of the 31st Annual Symposium on Applied Computing, Apr. 04-08,ACM, Pisa, Italy, pp: 1780-1783. DOI: 10.1145/2851613.2851908

Theis Geraldi, Ricardo & Oliveira Jr, Edson. 2017. Defect Types and Software Inspection Techniques: A Systematic Mapping Study. Journal of Computer Science. 13. 470-495. 10.3844/jcssp.2017.470.495.

Walia, G.S. and J.C. Carver, 2009. A systematic literature review to identify and classify software requirement errors. Inform. Software Technol., 51: 1087-1109. DOI: 10.1016/j.infsof.2009.01.004

Writing Views. 2018, May 27. Retrieved November 5, 2018, from https://docs.djangoproject.com/en/2.0/topics/http/views/